Desarrollo de software de aplicación

Niveles esperados de Desarrollo de software de aplicacion

Niveles de desempeño	
Satisfactorio	Sobresaliente
El sustentante con un nivel de desempeño Satisfactorio es capaz de interpretar las funciones de los manejadores de bases de datos y contrastar esta tecnología con la de *data warehouse*. Asimismo, es capaz de analizar las fases de la ingeniería de *software* y las bases de datos centralizadas y distribuidas. También puede valorar los elementos de la gestión de los proyectos de *software*, los paradigmas y lenguajes de programación, el modelo entidad-relación, la arquitectura de bases de datos distribuida y la seguridad informática para el desarrollo de *software* de aplicación.	Además de lo señalado en el nivel Satisfactorio, el sustentante con nivel Sobresaliente es capaz de analizar la concurrencia en bases de datos distribuidas y los mecanismos locales y remotos de seguridad, así como valorar aspectos de la usabilidad y de los diagramas UML, características de programación orientada a objetos (polimorfismo o herencia), programación funcional, programación de bases de datos (disparadores o procedimientos almacenados) y mecanismos de autenticación mediante algoritmos de cifrado (simétricos y asimétricos) en soluciones de *software*.

Subtemas

3.1. Ingeniería de software

3.1.1 Análisis de Sistemas de Software

Area 1

Área 1. Análisis de Sistemas de Software: (Todo con el libro de Somerville)

Procesos de Ing. de Requerimientos:

- Requerimientos funcionales
- Requerimientos no funcionales
- Requerimientos del dominio, del usuario y del sistema
- Especificación de la Interfaz
- Especificaciones de requerimientos de software (SRS)

Modelos de análisis del sistema:

- Contexto
- Comportamiento
- Datos
- Objetos

- Diagramas de UML (Casos de Uso)

Riesgos y su manejo

El análisis de sistemas de software es un componente crucial del desarrollo de software, y se centra en la identificación y especificación de los requisitos necesarios para un sistema. El libro de Ian Sommerville, "Software Engineering", es una referencia ampliamente reconocida en este campo y proporciona una cobertura detallada de los procesos de ingeniería de requerimientos. A continuación, se describen los conceptos clave que normalmente se abordan en este contexto:

1. **Requerimientos Funcionales**:
   - Son las capacidades específicas que un sistema de software debe ofrecer para satisfacer las necesidades del negocio y los usuarios.
   - Ejemplos comunes incluyen la gestión de transacciones, el procesamiento de datos y la generación de informes.
   - Estos requerimientos se enfocan en "qué debe hacer" el sistema.
2. **Requerimientos No Funcionales**:
   - Refieren a las cualidades y características del sistema que no están relacionadas directamente con las funciones específicas que realiza.
   - Incluyen aspectos como la confiabilidad (capacidad del sistema para funcionar sin fallos), la eficiencia (rendimiento del sistema bajo ciertas condiciones), y la usabilidad (facilidad con la que los usuarios pueden interactuar con el sistema).

- Estos requerimientos son cruciales para la satisfacción del usuario y la aceptación del sistema.

3. **Requerimientos del Dominio**:
   - Son los requerimientos específicos del dominio de aplicación del sistema.
   - Pueden incluir normas y reglamentaciones legales, terminología específica del dominio, y prácticas estándar de la industria.
   - Por ejemplo, en un sistema médico, los requerimientos del dominio podrían incluir el cumplimiento de normativas de salud y privacidad de datos.
4. **Requerimientos del Usuario**:
   - Son declaraciones generales y de alto nivel de lo que los usuarios esperan del sistema.
   - Estos requerimientos suelen ser más abstractos y pueden incluir la necesidad de mejorar la eficiencia del trabajo, facilitar ciertas tareas o mejorar la experiencia del usuario.
   - La comprensión de estos requerimientos es crucial para el diseño de un sistema que sea útil y satisfactorio para los usuarios finales.
5. **Requerimientos del Sistema**:
   - Proporcionan una descripción detallada y técnica de lo que el sistema debe hacer.
   - Incluyen especificaciones precisas de las funcionalidades del software, cómo debe interactuar con el hardware, y cómo debe responder a diferentes tipos de entradas.
   - Son fundamentales para los desarrolladores y sirven como una guía detallada para el diseño y la implementación del sistema.
6. **Especificación de la Interfaz**:
   - Se refiere a cómo el sistema interactuará con otros sistemas, dispositivos, y los usuarios.
   - Incluye la definición de interfaces de usuario (UI), interfaces de programación de aplicaciones (APIs), y protocolos de comunicación.
   - Una interfaz bien diseñada es esencial para asegurar la interoperabilidad y la facilidad de uso.
7. **Especificaciones de Requerimientos de Software (SRS)**:
   - Es un documento que contiene todos los requerimientos de un sistema de software.
   - Sirve como un contrato entre los desarrolladores y los clientes, estableciendo lo que el software hará y cómo se espera que lo haga.

- Un SRS bien elaborado reduce la ambigüedad, permite realizar estimaciones precisas y sirve como una base para futuras etapas de prueba y mantenimiento del software.

Es esencial que los ingenieros de software entiendan cómo recolectar, analizar y especificar claramente los requisitos para construir un sistema que satisfaga las necesidades de los usuarios y otras partes interesadas. Un SRS bien elaborado es fundamental para el éxito de un proyecto de software, ya que guía todas las fases subsecuentes del desarrollo y es a menudo utilizado para la validación y verificación del producto final.

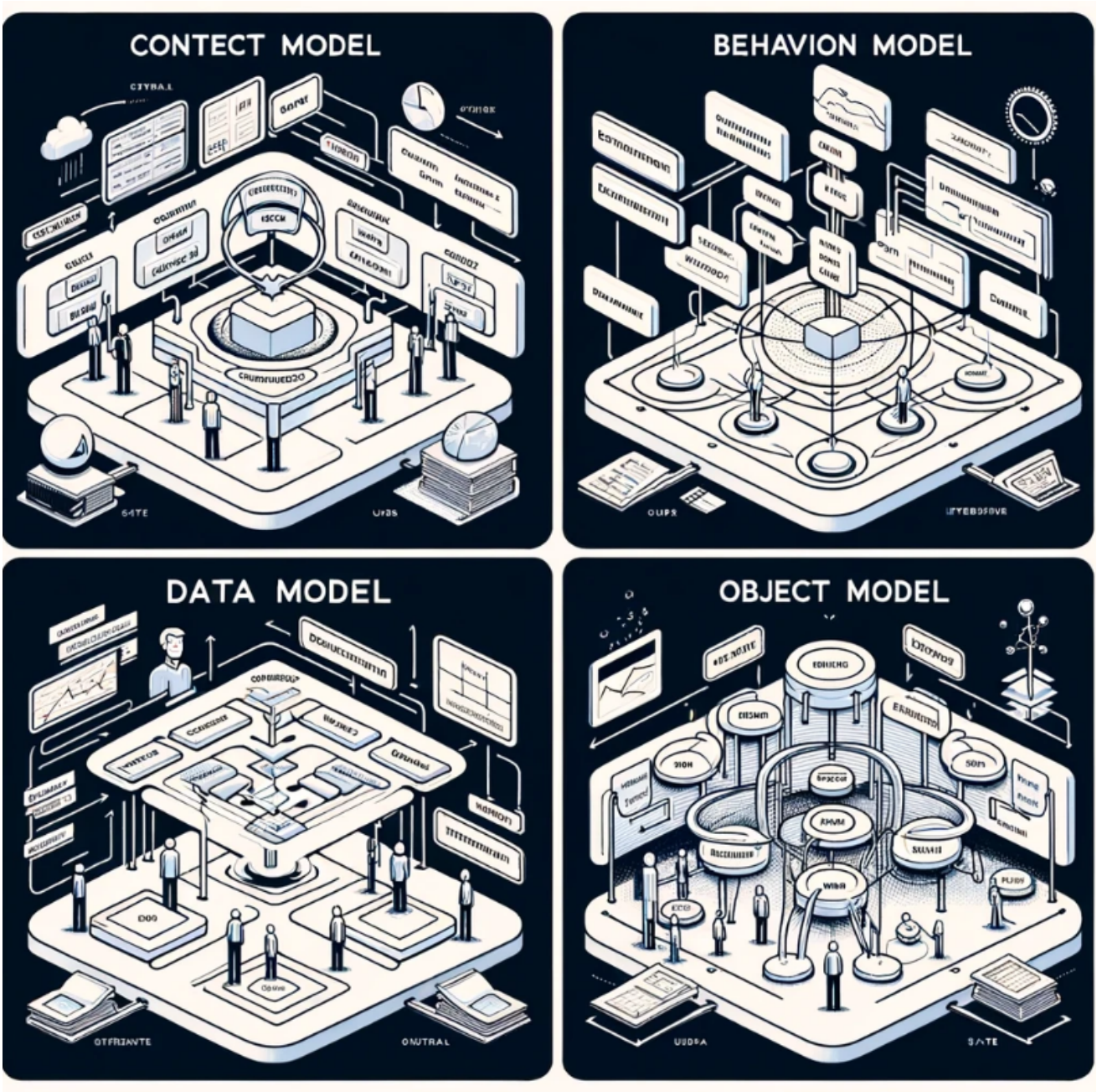
El análisis de sistemas es una parte vital del desarrollo de software, y los modelos que mencionas juegan un papel crucial en este proceso. Vamos a profundizar en cada uno de estos modelos para entender mejor cómo funcionan y su importancia:

1. **Modelo de Contexto**:
   - **Propósito**: Este modelo ayuda a entender el entorno operativo del sistema. Define cómo el sistema se ajusta en el entorno más amplio y su interacción con otros sistemas o entidades.
   - **Componentes Clave**:
     - **Entorno del Sistema**: Identifica los elementos externos con los que el sistema interactuará, como otros sistemas, usuarios y procesos de negocio.
     - **Interfaces**: Establece los puntos de interacción entre el sistema y su entorno, lo que ayuda a definir los límites del sistema.
   - **Importancia**: Esencial para asegurar que el sistema se integra adecuadamente en su entorno operativo y cumple con las expectativas de las partes interesadas externas.
2. **Modelo de Comportamiento**:
   - **Propósito**: Se enfoca en describir qué hace el sistema y cómo responde a ciertos eventos o entradas.
   - **Componentes Clave**:
     - **Diagramas de Estado**: Ilustran cómo el estado del sistema cambia en respuesta a eventos internos o externos.
     - **Funciones del Sistema**: Detalla las operaciones que el sistema debe realizar en respuesta a ciertas solicitudes o condiciones.
   - **Importancia**: Fundamental para comprender el flujo de trabajo del sistema y cómo debe comportarse en diferentes escenarios.

3. **Modelo de Datos**:
    - **Propósito**: Trata sobre la estructuración y organización de los datos dentro del sistema.
    - **Componentes Clave**:
        - **Diagramas de Entidad-Relación (ER)**: Muestran cómo diferentes entidades (como datos o conceptos) se relacionan entre sí dentro del sistema.
        - **Modelos de Datos de Objeto**: Utilizados en enfoques de modelado orientados a objetos, estos modelos describen la estructura de los datos en términos de objetos y sus relaciones.
    - **Importancia**: Clave para garantizar que los datos se manejen de manera eficiente y coherente, y que el sistema pueda almacenar, recuperar y procesar datos de manera efectiva.
4. **Modelo de Objetos**:
    - **Propósito**: Utilizado en el enfoque de programación orientada a objetos, este modelo analiza el sistema en términos de objetos y clases.
    - **Componentes Clave**:
        - **Clases de Objetos**: Definen las propiedades y comportamientos de los objetos dentro del sistema.
        - **Relaciones**: Describen cómo diferentes objetos o clases interactúan y se relacionan entre sí.
    - **Importancia**: Esencial para el diseño de sistemas en el paradigma de programación orientada a objetos, proporcionando una estructura clara y modular para el desarrollo del software.

Cada uno de estos modelos proporciona una perspectiva diferente pero complementaria del sistema. Su uso combinado permite a los desarrolladores y analistas comprender completamente tanto la estructura como el comportamiento del sistema, lo que es crucial para su diseño y desarrollo exitoso.

Aquí tienes visuales que representan cada uno de los modelos de análisis de sistemas:



1. **Modelo de Contexto**: Muestra un sistema en su entorno, ilustrando las entidades externas, usuarios e interfaces.
2. **Modelo de Comportamiento**: Presenta un diagrama de estado, mostrando cómo el sistema reacciona a diferentes eventos internos o externos.
3. **Modelo de Datos**: Exhibe un diagrama de entidad-relación (ER), destacando la estructura de datos dentro del sistema.
4. **Modelo de Objetos**: Visualiza un sistema con clases de objetos, sus atributos y las relaciones entre ellos.

Los Diagramas de UML (Lenguaje Unificado de Modelado)

Para Casos de Uso son una herramienta esencial en el desarrollo de software para representar los requerimientos funcionales de un sistema. Vamos a explorar más a fondo su propósito y cómo funcionan:

1. **Propósito de los Diagramas de Casos de Uso de UML**:
   - Estos diagramas se utilizan para describir cómo los diferentes actores (usuarios o sistemas) interactúan con el sistema para lograr objetivos específicos.
   - Proporcionan una vista de alto nivel de los requerimientos funcionales, mostrando las principales funcionalidades del sistema y quién las utiliza.
2. **Componentes Clave**:
   - **Actores**: Representan a los usuarios o sistemas que interactúan con el sistema. Pueden ser internos o externos al sistema.
   - **Casos de Uso**: Son las acciones o secuencias de eventos que el sistema realiza en respuesta a una interacción con un actor.
   - **Relaciones**: Las líneas o enlaces que conectan actores con casos de uso, mostrando las interacciones.
3. **Cómo Ayudan a Entender las Funcionalidades del Sistema**:
   - Al visualizar las interacciones entre los actores y el sistema, estos diagramas ayudan a comprender cómo los usuarios utilizarán el sistema.
   - Facilitan la identificación de las funcionalidades clave que el sistema debe desarrollar para satisfacer las necesidades de los usuarios.
   - Ayudan a los desarrolladores y partes interesadas a tener una comprensión común de lo que el sistema debe hacer.

La construcción de un diagrama UML (Lenguaje Unificado de Modelado) sigue una serie de pasos y reglas específicas para asegurar que el diagrama sea claro, preciso y útil. Aquí te describo cómo se construye un diagrama UML, enfocándome en los diagramas de casos de uso, que son comúnmente usados para representar los requerimientos funcionales de un sistema:

1. **Definir el Propósito del Diagrama**:
   - Antes de empezar, es esencial determinar qué aspecto del sistema se va a modelar. Esto podría ser un proceso, un sistema completo, o una parte específica del sistema.
2. **Identificar los Actores**:
   - Los actores son entidades externas que interactúan con el sistema. Pueden ser usuarios, otros sistemas, o procesos.

- Identifica todos los tipos de usuarios y sistemas que interactuarán con el sistema.

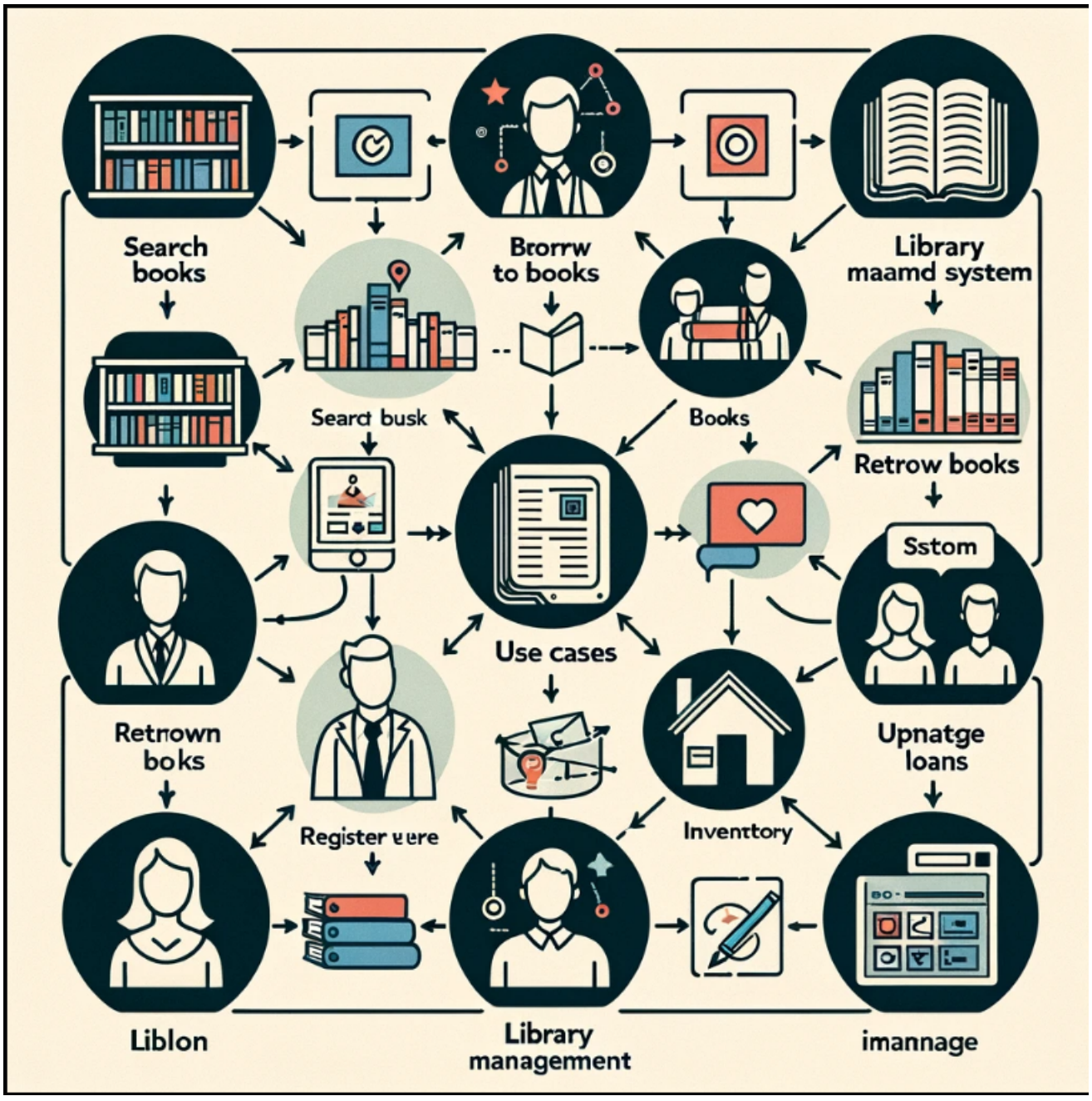
3. **Identificar los Casos de Uso**:
   - Los casos de uso son las funcionalidades o servicios que el sistema proporciona a los actores.
   - Enumera las funciones principales que el sistema debe realizar.
4. **Establecer Relaciones**:
   - Determina cómo los actores interactúan con los casos de uso.
   - Las relaciones se pueden clasificar como asociaciones (cuando un actor está involucrado en un caso de uso) o inclusiones/extensiones (cuando un caso de uso incluye o se extiende con otro).
5. **Crear el Diagrama**:
   - Usa un software de modelado UML o dibuja el diagrama manualmente.
   - Representa los actores como figuras de "palitos" y los casos de uso como óvalos.
   - Conecta los actores con los casos de uso relevantes mediante líneas.
6. **Añadir Detalles**:
   - Añade detalles a los casos de uso y actores si es necesario, como descripciones o restricciones.
   - Asegúrate de que el diagrama sea fácil de entender y evita sobrecargarlo con demasiada información.
7. **Revisar y Refinar**:
   - Revisa el diagrama para asegurarte de que todos los elementos importantes estén representados y las relaciones sean correctas.
   - Pide retroalimentación a las partes interesadas y haz ajustes según sea necesario.

Reglas Importantes para los Diagramas de UML:

- **Claridad**: El diagrama debe ser fácil de entender.
- **Consistencia**: Usa la misma notación y estilo a lo largo de todo el diagrama.
- **Relevancia**: Incluye solo la información que es importante para el propósito del diagrama.
- **Exactitud**: Asegúrate de que el diagrama refleje con precisión el sistema o proceso que estás modelando.

Al seguir estos pasos y reglas, puedes crear un diagrama de UML efectivo que sirva como una herramienta valiosa para la comprensión y el desarrollo del sistema.

Aquí tienes una representación visual de un Diagrama de Casos de Uso de UML para un sistema de biblioteca:



- **Actores**:
  - **Usuario de la Biblioteca**
  - **Bibliotecario**
  - **Sistema de Gestión de la Biblioteca**
- **Casos de Uso**:
  - **Buscar Libros**

    - **Prestar Libros**
    - **Devolver Libros**
    - **Registrar Usuarios**
    - **Actualizar Inventario**
    - **Gestionar Préstamos**
- **Interacciones**:
    - Las líneas conectan a los actores con los casos de uso relevantes, mostrando cómo interactúan con el sistema.

Este diagrama proporciona una visión clara de cómo los diferentes actores interactúan con un sistema de biblioteca, demostrando las funcionalidades clave que el sistema debe ofrecer. Los diagramas de UML como este son herramientas esenciales para visualizar las necesidades funcionales de un sistema y cómo se espera que los usuarios interactúen con él.

El manejo de riesgos

Es un aspecto fundamental en el desarrollo y análisis de sistemas de software. Veamos más detalladamente cada paso de este proceso y cómo los diagramas de UML pueden apoyar en estas actividades:

1. **Identificación de Riesgos**:
    - En este paso, se identifican todos los posibles riesgos que podrían afectar el proyecto. Esto incluye riesgos técnicos (como tecnología inadecuada o falta de habilidades técnicas), riesgos de proyecto (como plazos poco realistas o problemas de presupuesto), riesgos de negocio (como cambios en el mercado o en la demanda del cliente) y riesgos operativos (como fallas en el sistema o problemas de seguridad).
    - Los diagramas de UML pueden ayudar a identificar riesgos técnicos, especialmente aquellos relacionados con la arquitectura del sistema, la complejidad del diseño o las dependencias entre diferentes componentes del sistema.
2. **Análisis de Riesgos**:
    - Una vez identificados los riesgos, se analizan para entender su potencial impacto y probabilidad de ocurrencia. Esto ayuda a priorizar los riesgos en función de su severidad.
    - Los diagramas de casos de uso y los diagramas de secuencia de UML pueden ser útiles para analizar cómo los riesgos técnicos podrían afectar las

diferentes funcionalidades y procesos del sistema.

3. **Planificación de Respuesta a Riesgos**:
    - Para cada riesgo identificado y analizado, se desarrolla un plan de acción. Esto puede incluir evitar el riesgo, mitigarlo, transferirlo o aceptarlo.
    - Los diagramas de UML pueden ser utilizados para planificar respuestas a riesgos técnicos, como reestructurar el diseño del sistema o modificar los casos de uso para reducir la complejidad.
4. **Monitoreo de Riesgos**:
    - Finalmente, es esencial monitorear continuamente los riesgos a lo largo del proyecto para detectar cambios en su estado y asegurarse de que las estrategias de respuesta sean efectivas.
    - Los diagramas de componentes o de despliegue de UML pueden ser útiles para monitorear cómo se implementan las respuestas a los riesgos en la arquitectura y la infraestructura del sistema.

Para aplicar el manejo de riesgos al ejemplo del sistema de biblioteca que discutimos anteriormente, seguiríamos los pasos de identificación, análisis, planificación de respuesta y monitoreo de riesgos. Aquí te muestro cómo se podría hacer esto:

1. **Identificación de Riesgos**:
    - **Técnicos**: Fallos en el sistema de gestión de la biblioteca, como problemas en la base de datos o incompatibilidades de software.
    - **De Proyecto**: Retrasos en el desarrollo debido a una planificación inadecuada o falta de recursos.
    - **De Negocio**: Cambios en los requisitos del sistema debido a nuevas políticas de la biblioteca o expectativas cambiantes de los usuarios.
    - **Operativos**: Problemas de seguridad de los datos, como la pérdida de información de usuarios o de inventario de libros.
2. **Análisis de Riesgos**:
    - Evaluar la probabilidad y el impacto de cada riesgo. Por ejemplo, la pérdida de datos podría tener un impacto alto, mientras que los retrasos en el desarrollo pueden tener un impacto medio.
3. **Planificación de Respuesta a Riesgos**:
    - **Para Riesgos Técnicos**: Implementar protocolos de respaldo y recuperación de datos, y pruebas de compatibilidad de software.
    - **Para Riesgos de Proyecto**: Mejorar la planificación del proyecto y asegurar recursos adecuados.

- **Para Riesgos de Negocio**: Mantener una comunicación constante con las partes interesadas para adaptarse a los cambios en los requisitos.
- **Para Riesgos Operativos**: Implementar medidas de seguridad robustas para la protección de datos.

4. **Monitoreo de Riesgos**:
   - Establecer revisiones periódicas para evaluar el estado de los riesgos y la efectividad de las estrategias de respuesta.
   - Ajustar las estrategias según sea necesario basándose en el monitoreo continuo.

En este proceso, los diagramas de UML podrían usarse para ilustrar cómo los cambios en el sistema (como mejoras en la seguridad o en la gestión de datos) afectan la arquitectura general del sistema y los procesos de usuario. Por ejemplo, un diagrama de componentes podría usarse para planificar y visualizar cambios en la infraestructura de TI para mitigar los riesgos técnicos.

Este enfoque sistemático y proactivo hacia el manejo de riesgos es crucial para minimizar los problemas potenciales que podrían comprometer el éxito del proyecto del sistema de biblioteca.

3.1.2 Diseño de Sistemas de Software

Area 2

Área 2. Diseño de Sistemas de Software: (Con cualquier libro de UML de los 3 amigos y el libro de Somerville)

Diseño Arquitectónico
- Arquitecturas básicas de software
- Patrones básicos de arquitectura de software (MVC)

Diseño de Software
- Diagramas de UML (Clases, Objetos, Secuencia, Comunicación, Actividad y Estados)
- Patrones básicos de diseño de software.
  -Creacionales (Sigleton, Factory Method, Abstract factory)
  -Structural (Facade, Composite, Bridge)
  -Conductual (Iterator, visitor, Strategy)

Diseño de Interfaces de usuario

El diseño arquitectónico en el desarrollo de software es un proceso crucial que define la estructura subyacente de un sistema. Veamos más detalladamente los aspectos que has mencionado:

El diseño arquitectónico en el desarrollo de software es un proceso crucial que define la estructura subyacente de un sistema. Veamos más detalladamente los aspectos que has mencionado:

1. **Diseño Arquitectónico**:
   - **Definición**: Se refiere a la creación de un esquema de alto nivel que especifica la estructura general del software. Este diseño define cómo se organizan y se comunican los diferentes componentes del sistema.
   - **Importancia**: Un buen diseño arquitectónico es fundamental para garantizar que el sistema sea robusto, escalable, mantenible y eficiente.
2. **Arquitecturas Básicas de Software**:
   - **Arquitectura en Capas**: Divide el sistema en capas lógicas, cada una con una funcionalidad específica (por ejemplo, presentación, lógica de negocio, acceso a datos). Esto facilita la separación de preocupaciones y la reutilización del código.
   - **Cliente-Servidor**: En este modelo, las tareas se distribuyen entre proveedores de recursos o servicios (servidores) y solicitantes de servicios (clientes). Es común en aplicaciones web y de red.
   - **Basada en Eventos**: Esta arquitectura se centra en la respuesta a eventos, lo que la hace ideal para aplicaciones que necesitan manejar una gran cantidad de acciones asíncronas, como interfaces de usuario o sistemas de procesamiento de transacciones.
3. **Patrones Básicos de Arquitectura de Software**:
   El Modelo-Vista-Controlador (MVC)
   Es un patrón de diseño arquitectónico ampliamente utilizado en el desarrollo de software, especialmente en aplicaciones web. Este patrón se utiliza para separar las preocupaciones de una aplicación en tres componentes principales: modelo, vista y controlador. Aquí está una descripción detallada de cada componente y cómo interactúan entre sí:
4. **Modelo**:
   - **Descripción**: El modelo representa la lógica de negocio y los datos subyacentes de la aplicación. Gestiona el comportamiento y el estado de la aplicación, incluyendo los datos, las reglas de negocio, la lógica y las funciones.

- **Responsabilidades**: Encargado de acceder a la capa de almacenamiento de datos, definir las reglas de negocio, y manipular los datos. El modelo notifica a la vista cuando hay un cambio en su estado.
- **Ejemplo**: En una aplicación de comercio electrónico, el modelo manejaría datos como el inventario de productos, los precios y las descripciones.

5. **Vista**:
   - **Descripción**: La vista es la interfaz de usuario de la aplicación. Presenta los datos del modelo al usuario y muestra cualquier representación visual de esos datos.
   - **Responsabilidades**: Recibe los datos del modelo y los muestra al usuario. En aplicaciones web, la vista suele ser una página HTML generada dinámicamente.
   - **Ejemplo**: Las páginas que muestran la lista de productos y el carrito de compras en una tienda en línea.
6. **Controlador**:
   - **Descripción**: El controlador actúa como intermediario entre el modelo y la vista. Procesa todas las solicitudes de entrada del usuario, solicitando datos al modelo y devolviendo la vista apropiada.
   - **Responsabilidades**: Maneja la entrada del usuario, interactúa con el modelo y selecciona una vista para responder a la solicitud del usuario.
   - **Ejemplo**: En un sitio web de comercio electrónico, el controlador manejaría acciones como añadir un artículo al carrito, actualizar la cantidad de un artículo o realizar el checkout.

Interacción entre los Componentes:

- El usuario interactúa con la interfaz de usuario (vista).
- La vista envía las solicitudes de usuario al controlador.
- El controlador interpreta las solicitudes, manipula el modelo según sea necesario y determina la vista adecuada para responder.
- El modelo realiza las operaciones de lógica de negocio y notifica a la vista de los cambios de estado.
- La vista actualiza la interfaz de usuario según los cambios en el modelo.

El patrón MVC promueve una separación clara de responsabilidades, lo que facilita la mantenibilidad y escalabilidad de la aplicación, así como la posibilidad de tener múltiples vistas para un mismo modelo y una mejor organización del código.

El diseño arquitectónico es esencial para crear un marco sólido sobre el cual se construirá el software. Al utilizar patrones de diseño y arquitecturas bien establecidas, los desarrolladores pueden asegurarse de que su software sea capaz de adaptarse a cambios futuros, sea fácil de mantener y eficiente en su funcionamiento.

2. **Arquitecturas Básicas de Software**:
    - **Arquitectura en Capas**: Divide el sistema en capas lógicas, cada una con una funcionalidad específica (por ejemplo, presentación, lógica de negocio, acceso a datos). Esto facilita la separación de preocupaciones y la reutilización del código.
    - **Cliente-Servidor**: En este modelo, las tareas se distribuyen entre proveedores de recursos o servicios (servidores) y solicitantes de servicios (clientes). Es común en aplicaciones web y de red.
    - **Basada en Eventos**: Esta arquitectura se centra en la respuesta a eventos, lo que la hace ideal para aplicaciones que necesitan manejar una gran cantidad de acciones asíncronas, como interfaces de usuario o sistemas de procesamiento de transacciones.
3. **Patrones Básicos de Arquitectura de Software**:
    - **Modelo-Vista-Controlador (MVC)**: Es uno de los patrones de diseño más comunes, especialmente en aplicaciones web.
        - **Modelo**: Contiene la lógica de negocio y los datos del sistema.
        - **Vista**: Es la interfaz de usuario, que muestra los datos al usuario.
        - **Controlador**: Actúa como intermediario entre el modelo y la vista, manejando la lógica de control y las entradas del usuario.
    - **Ventajas de MVC**: Permite una clara separación de responsabilidades, lo que facilita el mantenimiento y la escalabilidad del sistema. También promueve la modularidad, permitiendo que equipos diferentes trabajen en la vista, el controlador y el modelo de manera independiente.

El diseño arquitectónico es esencial para crear un marco sólido sobre el cual se construirá el software. Al utilizar patrones de diseño y arquitecturas bien establecidas, los desarrolladores pueden asegurarse de que su software sea capaz de adaptarse a cambios futuros, sea fácil de mantener y eficiente en su funcionamiento.

El diseño de software es una fase esencial en el desarrollo de aplicaciones y sistemas. Utiliza una variedad de diagramas de UML (Lenguaje Unificado de Modelado) para representar visualmente y especificar la estructura y el comportamiento del software. Cada tipo de diagrama tiene un propósito específico:

1. **Diagramas de Clases**:
    - **Propósito**: Muestran la estructura estática del sistema.
    - **Componentes**: Incluyen clases, atributos, métodos y las relaciones entre ellas, como herencia, asociación, dependencia y agregación.
    - **Uso**: Fundamental en la fase de diseño para definir la arquitectura del software.
2. **Diagramas de Objetos**:
    - **Propósito**: Representan instancias de clases y sus relaciones en un momento específico.
    - **Uso**: Útiles para visualizar, documentar y probar la estructura del sistema y sus comportamientos en escenarios concretos.
3. **Diagramas de Secuencia y de Comunicación**:
    - **Diagramas de Secuencia**: Muestran la interacción de los objetos en función del tiempo. Destacan la secuencia de mensajes entre objetos.
    - **Diagramas de Comunicación**: Similar a los diagramas de secuencia pero se enfocan en el intercambio de mensajes y la organización de los objetos.
    - **Uso**: Ambos son cruciales para entender cómo se comunican los objetos durante la ejecución del sistema.
4. **Diagramas de Actividad**:
    - **Propósito**: Representan flujos de trabajo y procesos de negocio.
    - **Componentes**: Incluyen actividades, decisiones, paralelismo y sincronización.
    - **Uso**: Ayudan en la visualización de procesos operativos y lógica de negocio, especialmente útiles en la modelación de procesos complejos.
5. **Diagramas de Estados**:
    - **Propósito**: Ilustran cómo cambia el estado de un objeto en respuesta a eventos internos o externos.
    - **Componentes**: Estados, transiciones, eventos y actividades.
    - **Uso**: Importantes para modelar el comportamiento dinámico de los objetos, especialmente en sistemas donde el cambio de estado es una característica clave.

El diseño efectivo del software con estos diagramas facilita una mejor comprensión, desarrollo y mantenimiento del sistema. Permiten a los equipos visualizar la estructura y el comportamiento del sistema, asegurando que todos los aspectos del diseño estén bien pensados y alineados con los requerimientos del sistema.

Los patrones de diseño de software son soluciones reutilizables a problemas comunes en el diseño de software. Se dividen en tres categorías principales: creacionales, estructurales y conductuales. Cada categoría aborda diferentes aspectos del diseño y la arquitectura del software:

1. **Patrones Creacionales**:
   - **Singleton**: Asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a ella. Esto es útil para coordinar acciones en todo el sistema.
   - **Factory Method**: Define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar. Permite a una clase delegar la instanciación a subclases.
   - **Abstract Factory**: Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Esto permite que un sistema sea independiente de cómo se crean, componen y representan sus productos.
2. **Patrones Estructurales**:
   - **Facade**: Ofrece una interfaz unificada y simplificada a un conjunto de interfaces en un subsistema. Facade define una interfaz de nivel superior que facilita el uso del subsistema.
   - **Composite**: Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite a los clientes tratar de manera uniforme objetos individuales y composiciones.
   - **Bridge**: Desacopla una abstracción de su implementación para que ambas puedan variar de forma independiente. Esto es útil cuando se quiere evitar un enlace permanente entre una abstracción y su implementación.
3. **Patrones Conductuales**:
   - **Iterator**: Proporciona una forma de acceder a los elementos de una colección agregada sin exponer sus detalles subyacentes. Esto permite recorrer la colección de forma independiente de su implementación.
   - **Visitor**: Permite definir nuevas operaciones sin cambiar las clases de los elementos sobre los que opera. Es útil cuando se necesitan realizar operaciones diversas a través de una estructura de objetos compleja.
   - **Strategy**: Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. La estrategia permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

El uso de estos patrones en el diseño de software ayuda a solucionar problemas específicos de diseño, promueve el principio de reutilización y mejora la flexibilidad y la mantenibilidad del software. Al aplicar patrones de diseño, los desarrolladores pueden crear sistemas más robustos y escalables, facilitando el mantenimiento y la extensión del software en el futuro.
El diseño de interfaces de usuario es una parte esencial del desarrollo de software, ya que determina cómo los usuarios interactuarán con el sistema. Aquí te describo con más detalle este aspecto:

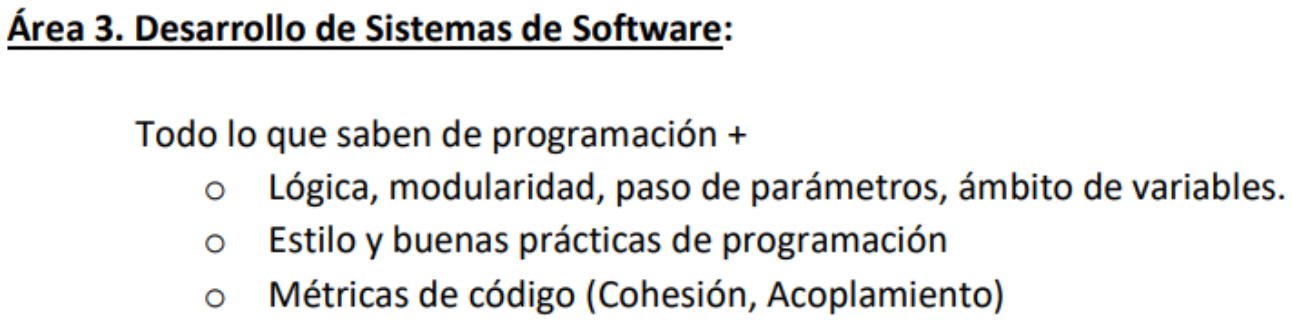
1. **Enfoque en la Interacción del Usuario**:
   - El objetivo principal es crear una interfaz que sea intuitiva y fácil de usar para los usuarios finales.
   - Implica entender cómo los usuarios interactuarán con el sistema y qué necesitan para realizar sus tareas de manera efectiva.
2. **Creación de Prototipos y Diseño de Pantallas**:
   - Los prototipos son versiones preliminares de una interfaz que se utilizan para explorar ideas y obtener retroalimentación de los usuarios. Pueden variar desde bocetos en papel hasta prototipos interactivos de alta fidelidad.
   - El diseño de pantallas involucra la disposición visual de los elementos en la interfaz, como botones, menús, campos de texto y otros controles interactivos.
3. **Elementos de Interacción**:
   - Incluyen todos los componentes a través de los cuales el usuario interactúa con el sistema, como formularios, controles de navegación, y elementos de entrada de datos.
   - Estos elementos deben ser diseñados pensando en la facilidad de uso y accesibilidad.
4. **Principios de Diseño de Interacción, Usabilidad y Experiencia de Usuario (UX)**:
   - **Diseño de Interacción**: Se centra en crear interfaces que faciliten la interacción del usuario con el sistema. Esto incluye la organización lógica de las funciones, una navegación clara y respuestas coherentes del sistema a las acciones del usuario.
   - **Usabilidad**: Se refiere a la facilidad con la que los usuarios pueden utilizar una interfaz para lograr sus objetivos. Implica simplicidad, claridad y un aprendizaje mínimo para el usuario.

- **Experiencia de Usuario (UX)**: Va más allá de la usabilidad para incluir cómo se siente un usuario al interactuar con el sistema. Un buen diseño de UX busca crear experiencias positivas, satisfactorias y atractivas para el usuario.

El diseño de interfaces de usuario es crucial para el éxito de cualquier sistema de software. Una interfaz bien diseñada no solo mejora la eficiencia y la productividad del usuario, sino que también puede influir en la percepción general del usuario sobre la calidad del sistema. Por lo tanto, es fundamental dedicar tiempo y recursos adecuados para asegurar que la interfaz del usuario sea atractiva, intuitiva y alineada con las necesidades y expectativas de los usuarios.

3.1.3 Desarrollo de Sistemas de Software

Area 3



El área de Desarrollo de Sistemas de Software descrita en la imagen se centra en varios conceptos fundamentales de la ingeniería de software y la programación:

1. **Lógica de programación**: Se refiere a la estructura y el proceso de razonamiento utilizado para resolver problemas a través de instrucciones ejecutables por una computadora. La lógica es la base sobre la cual se construye el código, permitiendo que se ejecuten algoritmos y se manipulen datos.
2. **Modularidad**: La modularidad es el diseño o la organización del software en componentes independientes y reutilizables llamados módulos. Estos módulos pueden ser desarrollados, probados y depurados de forma independiente, lo que aumenta la legibilidad del código y facilita su mantenimiento y actualización.
3. **Paso de parámetros**: Se trata de la técnica de transferir datos a funciones o procedimientos durante la ejecución de un programa. Existen diferentes métodos para pasar parámetros, como por valor, donde se pasa una copia del dato, o por

referencia, donde se pasa la dirección de memoria del dato, permitiendo que la función modifique el valor original.

4. **Ámbito de variables**: El ámbito (o scope) de una variable determina en qué partes del programa puede ser accedida o modificada. Por ejemplo, una variable global puede ser accesible desde cualquier parte del programa, mientras que una variable local solo puede ser accedida dentro de la función o bloque de código donde fue declarada.
5. **Estilo y buenas prácticas de programación**: Esto incluye la adhesión a convenciones de nombrado, el uso de comentarios descriptivos, la estructuración clara del código, y el seguimiento de principios de diseño como DRY (Don't Repeat Yourself) y KISS (Keep It Simple, Stupid). Adoptar buenas prácticas ayuda a asegurar que el código sea legible, mantenible y escalable.

Las métricas de cohesión y acoplamiento son fundamentales para evaluar la calidad del diseño de software. Ambas juegan un papel crucial en la determinación de la mantenibilidad y escalabilidad del código. Veamos más en detalle cada una de ellas:

1. **Cohesión**:
   - **Definición**: Mide qué tan estrechamente relacionadas están las responsabilidades de un módulo o componente de software.
   - **Alta Cohesión**: Implica que un módulo se enfoca en una sola tarea o área de funcionalidad. Por ejemplo, una clase que gestiona únicamente operaciones relacionadas con la base de datos muestra alta cohesión.
   - **Beneficios de Alta Cohesión**:
     - **Facilita la Comprensión**: Al estar el módulo centrado en una tarea, es más fácil de entender y mantener.
     - **Facilita la Reutilización**: Los módulos con alta cohesión pueden ser más fácilmente reutilizados en diferentes partes del software.
2. **Acoplamiento**:
   - **Definición**: Mide la interdependencia entre diferentes módulos o componentes de un sistema.
   - **Bajo Acoplamiento**: Significa que los módulos son relativamente independientes unos de otros. Por ejemplo, si un cambio en un módulo no requiere cambios en otro, se dice que están bajo acoplamiento.
   - **Beneficios de Bajo Acoplamiento**:
     - **Facilita el Mantenimiento**: Los cambios en un módulo tienen un impacto limitado en otros módulos.

- **Promueve la Flexibilidad**: Los módulos pueden ser cambiados o reemplazados con menor esfuerzo y riesgo.

Ejemplo Práctico:
Imagina un sistema de gestión de pedidos donde un módulo gestiona la entrada de pedidos y otro módulo gestiona el inventario. Si el módulo de entrada de pedidos puede operar independientemente del módulo de inventario (bajo acoplamiento), y si cada módulo se enfoca únicamente en sus respectivas tareas (alta cohesión), el sistema será más fácil de mantener y actualizar.

En resumen, la alta cohesión y el bajo acoplamiento son ideales en el diseño de software, ya que mejoran la claridad, la mantenibilidad y la flexibilidad del código. Estas métricas ayudan a los desarrolladores a crear software que no solo cumple con los requisitos funcionales, sino que también es robusto y adaptable a cambios futuros.

3.1.4 Gestión de Proyectos de Software

Area 4

Área 4. Gestión de Proyectos de Software: (Todo con el libro de Somerville)

Modelos de Proceso de Software
- Modelos clásicos: Cascada, iterativo, espiral, etc.
- Modelos Agiles: XP, Scrum, etc.

Estimación de costos de software
- Modelo algorítmico
- Puntos de función
- COCOMO

Gestión de Calidad
- Calidad de proceso y de producto
- Métricas básicas de producto
- Modelo de capacidad y madurez de procesos (CMM-I)

Pruebas de Software
- Del sistema
- De componentes
- Casos de prueba
- Pruebas de Caja Blanca, Caja Negra y Caja gris.

La gestión de proyectos de software requiere una comprensión profunda de los diferentes modelos de procesos de software para seleccionar el enfoque más adecuado para cada proyecto. A continuación, se detallan los modelos clásicos y ágiles:

1. **Modelos Clásicos**:
   - **Cascada**: Un enfoque lineal y secuencial donde cada fase (requisitos, diseño, implementación, verificación, mantenimiento) se completa antes de comenzar la siguiente. Es fácil de entender y gestionar, pero poco flexible ante los cambios.
   - **Iterativo**: Permite desarrollar versiones incrementales del software, refinándolo en cada iteración. Ofrece flexibilidad para incorporar cambios y mejorar continuamente el producto.
   - **Espiral**: Combina la naturaleza secuencial de la cascada con la iterativa, con un fuerte enfoque en la evaluación de riesgos. Cada ciclo de la espiral

incluye planificación, análisis de riesgos, ingeniería y evaluación del cliente.

2. **Modelos Ágiles**:
   - **Extreme Programming (XP)**: Fomenta la adaptabilidad y la entrega rápida de software de alta calidad. Sus prácticas clave incluyen la programación en parejas, desarrollo orientado a pruebas, integración continua y diseño simple. Se centra en mejorar la calidad del software y la capacidad de adaptarse a los cambios.
   - **Scrum**: Marco de trabajo iterativo e incremental que promueve la colaboración del equipo, la flexibilidad y la entrega de productos funcionales. Los sprints son ciclos de desarrollo cortos y el proceso involucra roles definidos como Scrum Master, Dueño de Producto y Equipo de Desarrollo.

Selección del Modelo Apropiado:

- La elección entre modelos clásicos y ágiles depende de varios factores como el tamaño y la complejidad del proyecto, la necesidad de flexibilidad, los requisitos del cliente y los plazos de entrega.
- Los proyectos que tienen requisitos bien definidos y poco probables de cambiar pueden beneficiarse del modelo de cascada.
- Los proyectos que requieren flexibilidad y adaptación constante a los cambios suelen ser más adecuados para los modelos ágiles.
- En algunos casos, se puede adoptar un enfoque híbrido, combinando elementos de ambos modelos para adaptarse mejor a las necesidades del proyecto.

La gestión efectiva de proyectos de software implica no solo la elección del modelo de proceso adecuado, sino también la adaptación continua del proceso a las realidades cambiantes del proyecto y del entorno empresarial.

La estimación de costos en la gestión de proyectos de software es vital para planificar y asignar los recursos adecuados. Los tres enfoques comunes para la estimación de costos de software que mencionas son el Modelo Algorítmico, los Puntos de Función y el Modelo de Costo Constructivo (COCOMO). Aquí te proporciono más detalles sobre cada uno:

1. **Modelo Algorítmico**:
   - **Descripción**: Utiliza fórmulas matemáticas y algoritmos para estimar el costo del desarrollo de software. Estos algoritmos consideran factores como el

tamaño del software, la complejidad, la experiencia del equipo y las tecnologías utilizadas.
    - **Ejemplo**: COCOMO es un modelo algorítmico ampliamente reconocido en esta categoría.
    - **Aplicación**: Adecuado para proyectos donde los parámetros y variables pueden ser cuantificados de manera precisa.
2. **Puntos de Función**:
    - **Descripción**: Mide la funcionalidad que el software proporcionará desde la perspectiva del usuario. Se basa en factores como entradas y salidas del usuario, consultas de usuario, archivos y complejidad de las interfaces.
    - **Uso**: Este enfoque es útil en proyectos donde la funcionalidad del usuario es un factor crítico y puede ser definida claramente. Es particularmente popular en proyectos que siguen metodologías ágiles.
3. **COCOMO (Constructive Cost Model)**:
    - **Descripción**: Un modelo algorítmico desarrollado por Barry Boehm que estima el costo, el esfuerzo y el tiempo de desarrollo basándose en el tamaño del software, medido en líneas de código (LOC) o puntos de función.
    - **Versiones**:
        - **Básica**: Proporciona una estimación rápida y áspera.
        - **Intermedia**: Incorpora más factores de proyecto.
        - **Detallada**: Incluye más detalles y considera una gama más amplia de factores.
    - **COCOMO II**: Una versión actualizada que se adapta mejor al desarrollo de software moderno.

Consideraciones Adicionales:

- **Datos Históricos**: Utilizar datos de proyectos anteriores puede mejorar la precisión de las estimaciones.
- **Retroalimentación Continua**: Ajustar las estimaciones a lo largo del proyecto a medida que se dispone de más información es crucial para mantener la precisión.
- **Combinación de Técnicas**: A menudo, una combinación de estos enfoques proporciona las estimaciones más precisas y realistas.

En resumen, la elección del método de estimación de costos depende de la naturaleza del proyecto, la información disponible y la experiencia previa. La capacidad de ajustar las estimaciones basándose en la retroalimentación y los datos

del rendimiento del proyecto es clave para la gestión efectiva del proyecto de software.

La gestión de calidad en la ingeniería de software es esencial para asegurar que tanto los procesos como el producto final cumplan con los estándares de calidad requeridos. Aquí te explico en detalle los conceptos que has mencionado:

1. **Calidad de Proceso y de Producto**:
   - **Calidad de Proceso**: Se centra en la eficacia y eficiencia de los procesos utilizados para desarrollar el software. Un buen proceso de calidad debería resultar consistentemente en un producto de alta calidad. Se evalúa comparando los procesos actuales con los planeados y las mejores prácticas de la industria.
   - **Calidad de Producto**: Atiende a las características del software finalizado, incluyendo funcionalidad, usabilidad, fiabilidad, rendimiento, mantenibilidad y portabilidad.
2. **Métricas Básicas de Producto**:
   - Estas métricas ayudan a evaluar aspectos cuantitativos y cualitativos del producto de software. Ejemplos incluyen:
     - **Complejidad Ciclomática**: Evalúa la complejidad del flujo de control del software.
     - **Líneas de Código (LOC)**: Mide el tamaño del software.
     - **Defectos por Línea de Código**: Indica la calidad del código basada en la frecuencia de errores.
     - **Tiempo de Respuesta y Procesamiento**: Mide el rendimiento del software.
     - **Tasa de Fallos**: Evalúa la fiabilidad del software.
3. **Modelo de Capacidad y Madurez de Procesos (CMMI)**:
   - **Descripción**: CMMI es un marco de referencia para la mejora de procesos que ayuda a las organizaciones a desarrollar prácticas eficaces de gestión de proyectos y procesos de software.
   - **Niveles de Madurez**: Varían desde el Nivel 1 (inicial) hasta el Nivel 5 (optimizado), reflejando una evolución desde procesos ad hoc hasta un enfoque de mejora continua.

- **Aplicación**: Las organizaciones utilizan CMMI para evaluar su capacidad actual y planificar mejoras en sus procesos para aumentar la eficiencia y calidad en la entrega de software.

La adopción de prácticas sólidas de gestión de calidad en los procesos de desarrollo de software no solo mejora la calidad del producto final, sino que también ayuda a las organizaciones a ser más predecibles y eficientes en sus prácticas de desarrollo. Esto es crucial para satisfacer las expectativas de los clientes y para el éxito a largo plazo de los proyectos de software.

Las pruebas de software son una parte esencial del desarrollo de software que ayuda a asegurar que el producto final cumple con los requisitos y funciona como se espera. Aquí hay un desglose de los conceptos mencionados:

1. **Pruebas de Software:**
   - **Pruebas del sistema:** Verifican que el sistema completo funciona según lo previsto. Se realiza después de las pruebas de componentes o módulos y se enfoca en la interacción entre los componentes y el sistema como un todo.
   - **Pruebas de componentes:** También conocidas como pruebas unitarias, se centran en piezas individuales de software para asegurar que cada una funciona correctamente de manera aislada.
   - **Casos de prueba:** Son condiciones específicas bajo las cuales se prueba el software para verificar el comportamiento y la funcionalidad esperados. Un caso de prueba incluirá entradas, procedimientos de ejecución y los resultados esperados que ayudarán a determinar si una característica del software está funcionando como se esperaba.
   - **Pruebas de Caja Blanca:** También conocidas como pruebas estructurales, se basan en el análisis interno del código fuente y su ejecución. Se centran en la estructura lógica del software y pueden usar la cobertura de código como métrica.
   - **Pruebas de Caja Negra:** No se fijan en la estructura interna del código y se basan solo en las especificaciones y requisitos. El tester solo conoce las entradas y salidas esperadas y verifica la funcionalidad sin mirar cómo se realiza internamente.
   - **Pruebas de Caja Gris:** Son una combinación de pruebas de caja negra y caja blanca. El tester tiene acceso parcial al código interno y utiliza esta

información para diseñar los casos de prueba.

2. **Verificación y Validación:**
    - **Verificación:** Se refiere al proceso de asegurar que el producto está siendo desarrollado de manera correcta y conforme a los requisitos y especificaciones previas durante todas las fases del ciclo de vida del desarrollo del software. Es el proceso de comprobar que el producto cumple con las especificaciones y el diseño.
    - **Validación:** Se refiere al proceso de asegurar que el producto final cumple con el propósito y las expectativas del usuario final. Es decir, se comprueba si el software cumple con los requisitos y resuelve el problema para el que fue diseñado.

La verificación y la validación son actividades complementarias que se utilizan durante el ciclo de vida del desarrollo del software para asegurar que el producto final sea de calidad, cumpla con los requisitos y sea fiable.

3.2. Lenguajes de programación

3.2.1 Paradigmas de Programación

Los paradigmas de programación ofrecen diferentes enfoques y estilos para el desarrollo de software, cada uno con sus propios principios y técnicas. Aquí te describo con más detalle estos paradigmas:

1. **Imperativo**:
    - **Enfoque**: Se centra en cómo se deben ejecutar las tareas, utilizando secuencias de comandos o instrucciones.
    - **Ejemplos de Lenguajes**: C, Java.
    - **Características**: Control explícito del flujo del programa.
2. **Declarativo**:
    - **Enfoque**: Se enfoca en qué tarea se debe realizar, sin especificar cómo se debe hacer.
    - **Subparadigmas**: Programación funcional y lógica.
    - **Características**: Especificación de la lógica del cómputo sin su control.
3. **Orientado a Objetos (OOP)**:
    - **Enfoque**: Basado en "objetos" que contienen datos (atributos) y código (métodos).

- **Ejemplos de Lenguajes**: Java, Python.
- **Características**: Encapsulación, herencia, polimorfismo.

4. **Funcional**:
   - **Enfoque**: Construcción de programas utilizando funciones matemáticas.
   - **Ejemplos de Lenguajes**: Haskell, Scala.
   - **Características**: Inmutabilidad, funciones de primera clase, expresiones lambda.
5. **Lógico**:
   - **Enfoque**: Basado en la lógica formal.
   - **Ejemplos de Lenguajes**: Prolog.
   - **Características**: Programación basada en reglas y hechos.
6. **Estructurado**:
   - **Enfoque**: Programación clara y eficiente con subrutinas, bloques de código y bucles.
   - **Ejemplos de Lenguajes**: C.
   - **Características**: Estructura de control clara, evita el "salto" de código.
7. **Procedural**:
   - **Enfoque**: Llamadas a procedimientos o subrutinas.
   - **Características**: Variables de estado y secuencia de operaciones.
8. **Concurrente**:
   - **Enfoque**: Ejecución simultánea de múltiples procesos o hilos.
   - **Características**: Gestión de procesos simultáneos, sincronización y comunicación entre procesos.
9. **Orientado a Aspectos**:
   - **Enfoque**: Separación de preocupaciones en el código, especialmente para inyección de comportamiento.
   - **Características**: Modularización de aspectos como logging, seguridad, que atraviesan múltiples componentes.
10. **Reactiva**:

- **Enfoque**: Programación con flujos de datos asincrónicos y propagación de cambios.
- **Características**: Ideal para sistemas interactivos y en tiempo real, maneja flujos de datos y cambios de estado.

Cada paradigma tiene sus propias fortalezas y limitaciones, y la elección de uno sobre otro depende de los requerimientos específicos del proyecto, del problema a resolver y, en algunos casos, de las preferencias personales o la experiencia del equipo de desarrollo. Comprender estos paradigmas es clave para los desarrolladores, ya que permite seleccionar el enfoque más adecuado para cada situación.

3.2.2 Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es un paradigma de programación que ofrece una forma de estructurar programas para que las propiedades y comportamientos estén agrupados en objetos individuales. Aquí te detallo los conceptos clave de la POO:

1. **Objetos**:
   - **Definición**: Son instancias de clases. Un objeto encapsula datos (atributos) y métodos (funciones o procedimientos) que operan sobre esos datos.
   - **Uso**: Representan entidades del mundo real o conceptual con estado (datos) y comportamiento (métodos).
2. **Clases**:
   - **Definición**: Son plantillas o "moldes" para crear objetos. Definen los atributos y métodos que compartirán los objetos de esa clase.
   - **Características**: Pueden incluir un constructor para inicializar nuevos objetos.
3. **Encapsulamiento**:
   - **Definición**: Es el principio de ocultar los detalles internos de los objetos y exponer solo lo que es necesario para el resto del sistema.
   - **Beneficios**: Reduce la complejidad y aumenta la seguridad del software.
4. **Herencia**:
   - **Definición**: Permite que una clase (subclase) herede propiedades y métodos de otra clase (superclase).
   - **Ventajas**: Promueve la reutilización de código y la jerarquía de clases.
5. **Polimorfismo**:
   - **Definición**: Significa "muchas formas". En POO, permite que objetos de diferentes clases sean tratados como objetos de una clase común.
   - **Aplicación**: Un mismo método puede actuar de manera diferente en diferentes clases.
6. **Abstracción**:

- **Definición**: Se enfoca en identificar las características y comportamientos esenciales de un objeto, ignorando los detalles no esenciales.
- **Importancia**: Facilita la gestión de la complejidad del software.

7. **Mensajes**:
   - **Definición**: En POO, los objetos interactúan mediante el envío y recepción de mensajes (invocaciones de métodos).
8. **Excepciones**:
   - **Definición**: Mecanismos para manejar errores y situaciones excepcionales de manera controlada.
   - **Uso**: Ayuda a mantener la estabilidad y seguridad del programa.
9. **Composición**:
   - **Definición**: Es una alternativa a la herencia donde las clases se construyen utilizando otras clases.
   - **Beneficios**: Ofrece una mayor flexibilidad y evita problemas comunes de la herencia.
10. **Interfaces y Clases Abstractas**:
    - **Definición**: Definen un "contrato" que las clases deben implementar sin proporcionar una implementación completa.
    - **Uso**: Ayuda a establecer un diseño claro y a mantener la coherencia en el comportamiento de las clases.
11. **Constructores y Destructores**:
    - **Definición**: Métodos especiales para inicializar y limpiar un objeto.
    - **Función**: Los constructores se utilizan para crear instancias de un objeto, y los destructores para liberar recursos cuando un objeto ya no se necesita.

La POO es ampliamente utilizada en el desarrollo de software por su capacidad para organizar y manejar complejidad, facilitar la reutilización de código y mejorar la mantenibilidad de las aplicaciones. Lenguajes como C++ y Java ofrecen soporte integral para la POO, haciendo que sea un paradigma esencial para los desarrolladores modernos.

3.2.3 Programación Funcional

La programación funcional es un paradigma que ofrece un enfoque único y poderoso para el desarrollo de software, particularmente en entornos donde la concurrencia y la

inmutabilidad son importantes. Aquí tienes una descripción detallada de los conceptos clave de este paradigma:

1. **Funciones Puras**:
   - **Definición**: Son funciones cuyo resultado depende únicamente de sus argumentos y no producen efectos secundarios (como modificar variables globales).
   - **Beneficios**: Facilitan el testing y el razonamiento sobre el código.
2. **Inmutabilidad**:
   - **Definición**: Los datos o variables no se modifican después de su creación. Cualquier cambio resulta en la creación de una nueva copia de los datos.
   - **Ventajas**: Mejora la seguridad en entornos concurrentes y reduce los errores relacionados con el cambio de estado.
3. **Funciones de Orden Superior**:
   - **Definición**: Son funciones que toman otras funciones como argumentos o devuelven funciones como resultado.
   - **Uso**: Común en operaciones como map, filter y reduce.
4. **Evaluación Perezosa**:
   - **Definición**: Las expresiones se evalúan solo cuando su valor es necesario.
   - **Beneficios**: Puede mejorar el rendimiento y permite estructuras de datos infinitas.
5. **Expresiones Lambda (Funciones Anónimas)**:
   - **Definición**: Funciones definidas sin un nombre, utilizadas para implementar operaciones que no necesitan ser nombradas.
   - **Aplicación**: Utilizadas en operaciones de orden superior o como callbacks.
6. **Recursión**:
   - **Definición**: Técnica donde una función se llama a sí misma para iterar sobre datos o resolver un problema.
   - **Uso**: Reemplaza los bucles tradicionales y es esencial en la programación funcional.
7. **Cierre o Clausuras (Closures)**:
   - **Definición**: Funciones que recuerdan el entorno en el que fueron creadas y pueden acceder a variables fuera de su ámbito actual.
   - **Importancia**: Permiten técnicas avanzadas de programación y mantenimiento de estado de manera controlada.
8. **Pattern Matching**:

- **Definición**: Mecanismo para verificar y descomponer estructuras de datos según un patrón.
- **Uso**: Facilita la manipulación y el acceso a los datos.

9. **Tipado Funcional**:
   - **Definición**: Uso de sistemas de tipos fuertes y a menudo inferencia de tipos en lenguajes funcionales.
   - **Beneficio**: Ayuda a prevenir errores y mejora la calidad del código.
10. **Transparencia Referencial**:
    - **Definición**: Propiedad por la cual una expresión puede ser sustituida por su valor sin afectar el comportamiento del programa.
    - **Ventaja**: Facilita el razonamiento sobre el código y su refactorización.

La programación funcional se aplica en varios lenguajes y está ganando popularidad debido a su capacidad para manejar problemas complejos de manera elegante, especialmente en áreas como la programación concurrente y reactiva. Aunque algunos lenguajes están diseñados específicamente para ser funcionales, como Haskell y Erlang, otros lenguajes más comunes están adoptando características de la programación funcional, lo que demuestra su creciente influencia en el mundo del desarrollo de software.

3.2.4 Programación Web y Móvil

La programación web y móvil son áreas esenciales en el desarrollo de software moderno, cada una con su propio conjunto de tecnologías, herramientas y prácticas. Aquí te explico con más detalle estos dos campos:

Programación Web:

1. **Front-End**:
   - **Tecnologías**: HTML para estructura, CSS para estilo y JavaScript para interactividad.
   - **Frameworks y Bibliotecas**: React, Angular y Vue.js son populares para crear interfaces de usuario dinámicas y reactivas.
   - **Foco**: Diseño de interfaces de usuario y experiencia de usuario, accesibilidad, optimización de rendimiento.
2. **Back-End**:

- **Lógica del Servidor y Bases de Datos**: Gestiona la lógica del negocio, almacenamiento de datos y seguridad.
- **Lenguajes y Tecnologías**: PHP, MySQL, Java, Python, Ruby, Node.js.
- **Foco**: Arquitectura de aplicaciones, gestión de bases de datos, seguridad y escalabilidad.

3. **Intercambio de Información**:
   - **APIs REST**: Método estándar para la comunicación entre el front-end y el back-end, utilizando protocolos HTTP.

Programación Móvil:

1. **Android**:
   - **Lenguajes**: Principalmente Java y Kotlin.
   - **Herramientas**: Android Studio y Android SDK.
   - **Aspectos Clave**: Gestión del ciclo de vida de la aplicación, adaptabilidad a diferentes dispositivos y tamaños de pantalla.
2. **iOS**:
   - **Lenguajes**: Objective-C y Swift.
   - **Entorno de Desarrollo**: Xcode.
   - **Aspectos Clave**: Diseño centrado en el usuario, integración con el ecosistema de Apple.
3. **Desarrollo Multiplataforma**:
   - **Frameworks**: React Native, Flutter.
   - **Ventajas**: Permiten escribir un conjunto de código para múltiples plataformas, reduciendo tiempo y costos.

Tendencias y Mejores Prácticas:

- **Desarrollo Responsive**: Crear aplicaciones web que se ajusten a diferentes tamaños de pantalla.
- **Accesibilidad**: Asegurar que las aplicaciones sean accesibles para todos los usuarios, incluyendo aquellos con discapacidades.
- **Seguridad**: Implementar prácticas de seguridad para proteger datos y privacidad del usuario.
- **Aplicaciones Web Progresivas (PWA)**: Combinan las ventajas de las aplicaciones web y móviles, ofreciendo una experiencia de usuario similar a una aplicación móvil pero a través de un navegador.

La programación web y móvil requiere no solo conocimientos técnicos específicos de cada plataforma, sino también una comprensión de los principios generales del desarrollo de software, como la arquitectura de aplicaciones, el diseño de algoritmos y estructuras de datos, y la gestión del ciclo de vida del desarrollo de software.

3.3. Bases de datos

Conceptos de Bases de Datos:

Es importante recordar los conceptos generales de las bases de datos, el modelo de bases de datos y sus características. Esto lo pueden consultar en el Capítulo 2 del libro de Date, C. J.

Modelo Entidad-Relación:

Pueden consultar el Capítulo 2 del libro de Silberschatz, Korth, Sudarshan, o los Capítulos 3 y 4 del libro de Ramez Elmasri, Shamkant B. Navathe.

Modelo Relacional:

Pueden consultar el Capítulo 3 del libro de Silberschatz, Korth, Sudarshan, o los Capítulos 5 y 6 del libro de Ramez Elmasri, Shamkant B. Navathe.

Disparadores (Triggers):

Pueden consultar el Capítulo 8 del libro de Ramez Elmasri, Shamkant B. Navathe (Tema: 8.7).

Consultas en SQL:

Hay que recordar las consultas básicas como avanzadas con SQL. Pueden consultar el Capítulo 8 del libro de Ramez Elmasri, Shamkant B. Navathe (Temas: 8.4 y 8.5).

http://deletesql.com/viewtopic.php?f=5&t=5

Manejadores de Bases de Datos:

Hay que recordar los conceptos básicos de los manejadores, sus tipos, funciones, etc.

https://www.uv.mx/personal/ermeneses/files/2019/02/Clase1-AntecedentesSMBD.pdf

https://blog.powerdata.es/el-valor-de-la-gestion-de-datos/bid/406547/tipos-y-funci-n-de-los-gestores-de-bases-de-datos

https://aprender-libre.com/funciones-de-sistemas-manejadores-de-bases-de-datos/

Bases de Datos Distribuidas:

Este tema ha adquirido relevancia en los últimos años por la forma en que ahora se encuentran los sistemas. Es posible que este tema se pregunte porque también está la subárea de 4.3 Cómputo Distribuido.

3.3.1 Conceptos de Bases de Datos

Los conceptos de bases de datos son fundamentales para entender cómo se almacena, organiza y recupera la información en sistemas informáticos. Basándonos en el libro de C. J. Date y en principios generales de bases de datos, aquí te presento un resumen de algunos conceptos clave:

1. **Datos vs. Información**:
   - **Datos**: Hechos crudos almacenados en la base de datos.
   - **Información**: Datos procesados que han sido organizados y contextualizados.
2. **Modelos de Bases de Datos**:
   - **Modelo Relacional**: Utiliza tablas para representar y relacionar datos, basado en la teoría de conjuntos.

- **Modelo de Entidad-Relación**: Enfocado en entidades y sus relaciones, útil para el diseño de bases de datos.
- **Modelo Orientado a Objetos**: Modela datos como objetos, similar a la POO.
- **Modelo NoSQL**: Incluye modelos basados en documentos, clave-valor, grafos y columnas, ofreciendo flexibilidad en comparación con el modelo relacional.

3. **Características de las Bases de Datos**:
   - **Integridad**: Asegura la precisión y consistencia de los datos.
   - **Seguridad**: Protege los datos de accesos no autorizados o corrupción.
   - **Concurrencia**: Maneja el acceso simultáneo a los datos por múltiples usuarios.
   - **Recuperación**: Permite restaurar la base de datos después de fallos.
4. **Lenguajes de Bases de Datos**:
   - **SQL (Structured Query Language)**: Lenguaje estándar para manejar bases de datos relacionales.
5. **Sistemas de Gestión de Bases de Datos (SGBD)**:
   - Software que facilita la gestión de bases de datos, como MySQL, PostgreSQL, Oracle y SQL Server.
6. **Normalización**:
   - Proceso de diseño de bases de datos para reducir la redundancia y mejorar la dependencia de los datos.
7. **Índices**:
   - Estructuras que mejoran la velocidad de recuperación de datos en las bases de datos.
8. **Transacciones**:
   - Conjuntos de operaciones que se tratan como una unidad atómica de trabajo, siguiendo las propiedades ACID.

Estos conceptos proporcionan una base sólida para entender las bases de datos y son esenciales para cualquiera que trabaje en el campo de la informática y la gestión de datos. El enfoque teórico basado en la teoría de conjuntos y la lógica formal, como se detalla en el libro de C. J. Date, es crucial para una comprensión profunda de las bases de datos, especialmente las relacionales.

La diferencia entre el **Modelo Entidad-Relación (E-R) y el Modelo Relacional** radica en su propósito y enfoque en el diseño y la gestión de bases de datos. Ambos son

fundamentales en el campo de las bases de datos, pero se utilizan en diferentes etapas del diseño y tienen diferentes objetivos:

Modelo Entidad-Relación (E-R):

1. **Propósito y Enfoque**:
   - El Modelo E-R se utiliza principalmente en la fase de diseño conceptual de una base de datos.
   - Se centra en representar los datos como entidades (objetos) y las relaciones entre estas entidades.
2. **Componentes**:
   - Entidades que representan objetos o conceptos del mundo real (como "Empleado" o "Departamento").
   - Relaciones que representan cómo se asocian las entidades entre sí (como "trabaja en").
   - Atributos que definen las propiedades de las entidades y relaciones.
3. **Uso**:
   - Sirve para visualizar la estructura de datos de alto nivel y sus interconexiones.
   - Ayuda a entender el negocio y sus requerimientos de datos antes de implementar la base de datos.
4. **Representación**:
   - Se representa comúnmente mediante diagramas E-R, que son esquemas visuales de las entidades, sus atributos y relaciones.

Modelo Relacional:

1. **Propósito y Enfoque**:
   - El Modelo Relacional se utiliza en la fase de implementación de una base de datos.
   - Se centra en cómo se almacenan los datos, utilizando tablas (relaciones).
2. **Componentes**:
   - Tablas (o relaciones) que representan conjuntos de datos.
   - Cada fila en una tabla es una tupla o registro, y cada columna es un atributo.
   - Claves primarias y foráneas que establecen relaciones entre tablas.
3. **Uso**:
   - Utilizado para la implementación física de la base de datos.

- Sirve para realizar consultas y manipular los datos almacenados en la base de datos.

4. **Representación**:
   - Se representa mediante esquemas de base de datos que incluyen tablas, sus atributos y las relaciones (mediante claves) entre ellas.

Diferencias Clave:

- El Modelo E-R es más abstracto y se utiliza para el diseño conceptual, enfocándose en cómo se relacionan los elementos del dominio del problema.
- El Modelo Relacional es más concreto y se utiliza para la implementación y gestión de la base de datos, enfocándose en cómo se almacenan y se accede a los datos.

En resumen, mientras que el Modelo E-R ayuda en el diseño conceptual y en la representación de las relaciones entre entidades del mundo real, el Modelo Relacional se ocupa de cómo estos datos y relaciones se representan y almacenan efectivamente en una base de datos.

3.3.2 Modelo Entidad-Relación

El Modelo Entidad-Relación (E-R) es una herramienta esencial para el diseño conceptual de bases de datos. Te presento un ejemplo para ilustrar cómo se aplican los conceptos del modelo E-R en un escenario real:

Ejemplo: Sistema de Gestión de Biblioteca

1. **Entidades y Conjuntos de Entidades**:
   - **Entidades**: Libro, Autor, Usuario (lector).
   - **Conjuntos de Entidades**: Todos los Libros, todos los Autores, todos los Usuarios.
2. **Atributos**:
   - **Libro**: ID del Libro, Título, Año de Publicación.
   - **Autor**: ID del Autor, Nombre, Nacionalidad.
   - **Usuario**: ID del Usuario, Nombre, Dirección.
3. **Relaciones y Conjuntos de Relaciones**:
   - **Relación "escrito por"**: Entre Libro y Autor.
   - **Relación "prestado a"**: Entre Libro y Usuario.

4. **Llaves**:
    - **Llave Primaria (PK)**: ID del Libro, ID del Autor, ID del Usuario.
    - **Llave Foránea (FK)**: En la relación "escrito por", ID del Autor en la tabla Libro. En la relación "prestado a", ID del Usuario en la tabla Libro.
5. **Restricciones de Integridad**:
    - **Integridad de Entidad**: Cada libro, autor y usuario tiene una identificación única.
    - **Integridad Referencial**: Los ID de los autores y usuarios en las relaciones deben existir en sus respectivas tablas.
6. **Cardinalidad y Participación de las Relaciones**:
    - **Relación "escrito por"**: Un libro puede ser escrito por varios autores (cardinalidad muchos a muchos).
    - **Relación "prestado a"**: Un libro puede ser prestado a un usuario a la vez (cardinalidad uno a uno).
7. **Diagramas E-R**:
    - Representación gráfica de las entidades (Libro, Autor, Usuario) y sus relaciones ("escrito por", "prestado a").
8. **Modelado Avanzado de E-R**:
    - **Atributos Compuestos**: Dirección del Usuario (calle, ciudad, código postal).
    - **Atributos Multivalorados**: Diversos géneros para un Libro.
    - **Entidades Débiles**: Ejemplo no aplicable en este escenario.
9. **Especialización y Generalización**:
    - **Generalización**: 'Persona' como superclase generalizada de 'Autor' y 'Usuario'.
10. **Agregación**:
    - Ejemplo no aplicable en este escenario.

En este ejemplo, se utiliza el Modelo E-R para representar cómo se organiza y se relaciona la información en un sistema de gestión de biblioteca, mostrando la estructura de datos en términos de entidades, atributos y relaciones. Los diagramas E-R resultantes serían herramientas visuales efectivas para entender y comunicar el diseño de la base de datos de la biblioteca.

3.3.3 Modelo Relacional

El Modelo Relacional es una forma fundamental de organizar y acceder a los datos en bases de datos. Te presento un ejemplo sencillo para ilustrar los conceptos clave del modelo relacional, basado en una biblioteca:

Ejemplo: Sistema de Gestión de Biblioteca en un Modelo Relacional

1. **Estructura de una Base de Datos Relacional**:
    - **Relación o Tabla "Libros"**:
        - Tuplas: Cada fila representa un libro.
        - Atributos: ID del Libro, Título, Autor, Año de Publicación.
    - **Relación o Tabla "Usuarios"**:
        - Tuplas: Cada fila representa un usuario.
        - Atributos: ID del Usuario, Nombre, Dirección.
    - **Relación o Tabla "Préstamos"**:
        - Tuplas: Cada fila representa un préstamo.
        - Atributos: ID del Préstamo, ID del Libro, ID del Usuario, Fecha de Préstamo.
2. **Esquema de Relación**:
    - "Libros" (ID del Libro, Título, Autor, Año de Publicación)
    - "Usuarios" (ID del Usuario, Nombre, Dirección)
    - "Préstamos" (ID del Préstamo, ID del Libro, ID del Usuario, Fecha de Préstamo)
3. **Clave Primaria**:
    - "Libros": ID del Libro
    - "Usuarios": ID del Usuario
    - "Préstamos": ID del Préstamo
4. **Integridad de Relación**:
    - **Integridad de Entidad**: Ningún ID (clave primaria) en cualquier tabla puede ser nulo.
    - **Integridad Referencial**: Los IDs de libros y usuarios en la tabla "Préstamos" deben existir en las tablas "Libros" y "Usuarios".
5. **Operaciones del Álgebra Relacional**:
    - Ejemplo: Selección (encontrar todos los libros de un autor específico), Unión (combinar listas de libros y usuarios).
6. **Restricciones de Dominio**:
    - Año de Publicación debe ser un año válido.

7. **SQL**:
    - Utilizado para realizar consultas como "SELECT * FROM Libros WHERE Autor = 'Autor X'".
8. **Normalización**:
    - Las tablas están diseñadas para minimizar la redundancia (por ejemplo, los detalles del autor no se repiten en la tabla de libros).
9. **Formas Normales**:
    - Asegurando que las tablas cumplen con 1NF, 2NF, 3NF para mejorar la integridad y eficiencia.
10. **Vistas**:
    - Creación de una vista "LibrosDisponibles" que muestra libros que no están actualmente prestados.

Este ejemplo ilustra cómo se pueden aplicar los principios del modelo relacional para organizar eficientemente los datos en una base de datos de una biblioteca, facilitando el acceso y la gestión de los mismos.

3.3.4 Disparadores (Triggers)

Los disparadores, o "triggers", son una herramienta poderosa en las bases de datos para automatizar la respuesta a ciertos eventos. Te proporcionaré una explicación detallada de los conceptos clave de los triggers, acompañada de ejemplos prácticos:

1. **Conceptos Básicos de Triggers**:
    - **Definición**: Un trigger es un tipo de procedimiento almacenado que se activa automáticamente en respuesta a ciertos eventos en una tabla o vista, como inserciones, actualizaciones o eliminaciones.
    - **Uso**: Se utilizan para mantener la integridad de la base de datos, realizar cálculos automáticos, actualizar tablas relacionadas, etc.
2. **Eventos de Base de Datos**:
    - **Ejemplo**: Un trigger puede ser activado por una inserción en una tabla "Pedidos". Cada vez que se inserta un nuevo pedido, el trigger puede verificar si hay suficiente stock en la tabla "Inventario".
3. **Acciones del Trigger**:
    - **Ejemplo**: Si un pedido reduce el inventario por debajo de un cierto nivel, el trigger podría insertar automáticamente un registro en una tabla "PedidosDeReabastecimiento".

4. **Granularidad de los Triggers**:
    - **Triggers de Fila**: Se ejecutan para cada fila afectada por la operación.
    - **Triggers de Instrucción**: Se ejecutan una vez por operación de comando SQL.
    - **Ejemplo**: Un trigger de fila en la tabla "Empleados" que actualiza la edad de cada empleado modificado.
5. **Condición de Activación**:
    - **Ejemplo**: Un trigger puede activarse solo si el total de un pedido supera un cierto monto.
6. **Momento de Activación**:
    - **BEFORE**: Ejecuta el trigger antes de la operación de base de datos.
    - **AFTER**: Ejecuta el trigger después de la operación de base de datos.
    - **Ejemplo**: Un trigger AFTER en la tabla "Pedidos" para enviar una notificación una vez que se ha realizado un pedido.
7. **Variables de Transición y Tablas**:
    - **Ejemplo**: Acceder a los valores antiguos y nuevos de un salario en un trigger que responde a cambios en la tabla "Empleados".
8. **Consideraciones de Diseño y Rendimiento**:
    - Es crucial diseñar triggers que no causen cascadas de ejecuciones o ciclos infinitos, y que no afecten negativamente el rendimiento de la base de datos.
9. **Administración de Triggers**:
    - Incluye cómo crear, modificar y eliminar triggers usando SQL.
10. **Ejemplos de Triggers**:
    - **Ejemplo de Creación de un Trigger**:

```
CREATE TRIGGER verificar_stock
AFTER INSERT ON Pedidos
FOR EACH ROW
BEGIN
   -- Cuerpo del trigger para verificar y actualizar el inventario.
END;
```

Este ejemplo ilustra cómo se pueden utilizar los triggers para automatizar respuestas a eventos en bases de datos, lo que ayuda a mantener la integridad, ejecutar tareas relacionadas y gestionar reglas de negocio complejas de manera eficiente.

3.3.5 Consultas en SQL

Fundamentales para la manipulación y el acceso a los datos almacenados en bases de datos relacionales. El capítulo 8 de Ramez Elmasri y Shamkant B. Navathe abordará probablemente los siguientes temas:

1. **Sintaxis Básica de Consultas SQL:**
   - Uso de la instrucción `SELECT` para recuperar datos.
   - Selección de columnas.
   - Especificación de condiciones con `WHERE`.
   - Ordenación de resultados con `ORDER BY`.
2. **Funciones de Agregación y Agrupamiento:**
   - Uso de funciones como `COUNT`, `SUM`, `AVG`, `MIN`, y `MAX`.
   - Agrupamiento de resultados con `GROUP BY`.
   - Filtrado de grupos con `HAVING`.
3. **Consultas Avanzadas:**
   - Subconsultas y consultas correlacionadas.
   - Uso de `JOIN` para combinar filas de dos o más tablas.
   - Consultas con múltiples tablas y diferentes tipos de `JOIN` (INNER, LEFT, RIGHT, FULL OUTER).
4. **Consultas con Datos Temporales:**
   - Uso de `WITH` y Common Table Expressions (CTEs).
5. **Operaciones de Conjunto:**
   - Combinación de resultados de múltiples consultas con `UNION`, `INTERSECT`, y `EXCEPT`.
6. **Manipulación de Datos:**
   - Instrucciones `INSERT`, `UPDATE`, y `DELETE` para añadir, modificar y eliminar datos.
7. **Aspectos de Rendimiento de Consultas:**
   - Estrategias para optimizar consultas y mejorar la eficiencia.
8. **Ejemplos Prácticos y Casos de Uso:**

Las consultas en SQL son cruciales para interactuar con bases de datos relacionales. Aquí te muestro ejemplos prácticos de cada tipo de consulta mencionada:

1. **Sintaxis Básica de Consultas SQL**:

```
SELECT nombre, edad FROM Empleados WHERE edad > 30 ORDER BY edad DESC;
```

2. **Funciones de Agregación y Agrupamiento**:

```
SELECT departamento, COUNT(*) FROM Empleados GROUP BY departamento
HAVING COUNT(*) > 5;
```

3. **Consultas Avanzadas**:
   - **Subconsulta**:

```
SELECT nombre FROM Empleados WHERE id IN (SELECT empleado_id FROM
Proyectos WHERE proyecto = 'Alpha');
```

   - **JOIN**:

```
SELECT Empleados.nombre, Departamentos.nombre FROM Empleados INNER
JOIN Departamentos ON Empleados.departamento_id = Departamentos.id;
```

4. **Consultas con Datos Temporales**:

```
WITH TemporalCTE AS (SELECT nombre, fecha_ingreso FROM Empleados)
SELECT nombre FROM TemporalCTE WHERE fecha_ingreso BETWEEN '2020-01-01'
AND '2020-12-31';
```

5. **Operaciones de Conjunto**:

```
SELECT nombre FROM Empleados WHERE departamento = 'Ventas' UNION SELECT
nombre FROM Empleados WHERE departamento = 'Marketing';
```

6. **Manipulación de Datos**:
   - **INSERT**:

```
INSERT INTO Empleados (nombre, edad, departamento) VALUES ('Ana',
28, 'Ventas');
```

   - **UPDATE**:

```
UPDATE Empleados SET edad = 29 WHERE nombre = 'Ana';
```

- **DELETE**:

```
DELETE FROM Empleados WHERE nombre = 'Ana';
```

7. **Aspectos de Rendimiento de Consultas**:
   - Optimizar una consulta usando índices, por ejemplo, creando un índice en la columna `departamento`.
8. **Ejemplos Prácticos y Casos de Uso**:
   - Crear una consulta para obtener el total de ventas por empleado en un determinado año:

```
SELECT empleado_id, SUM(ventas) FROM Ventas WHERE año = 2021 GROUP BY empleado_id;
```

Estos ejemplos cubren una amplia gama de operaciones típicas en SQL y proporcionan una base sólida para entender cómo interactuar con bases de datos relacionales. Practicar con estos ejemplos y variaciones de ellos es una excelente manera de profundizar tus habilidades en SQL.

3.3.6 Manejadores de Bases de Datos

Los manejadores de bases de datos, también conocidos como Sistemas de Gestión de Bases de Datos (SGBD), son software esenciales en el mundo de la informática y las bases de datos. Aquí te presento un resumen de los puntos clave relacionados con los SGBD:

1. **Conceptos Básicos de los SGBD**:
   - **Definiciones y Objetivos**: Los SGBD son sistemas que facilitan la creación, manipulación y gestión de bases de datos. Permiten a los usuarios y aplicaciones interactuar con bases de datos de manera eficiente y segura.
   - **Beneficios**: Ofrecen una capa de abstracción para manejar datos, lo que ayuda en la organización, seguridad, respaldo y recuperación de datos.
2. **Tipos de SGBD**:

- **Basados en Modelos de Datos**: Incluyen SGBD relacionales (como MySQL, SQL Server), no relacionales (como MongoDB, Redis), jerárquicos y en red.
- **Sistemas de Bases de Datos Distribuidos**: Gestionan bases de datos distribuidas físicamente en múltiples ubicaciones.
- **Sistemas de Bases de Datos en Memoria**: Utilizan la memoria principal para almacenamiento de datos, ofreciendo un rendimiento más rápido.

3. **Funciones Principales**:
   - **Definición de Datos**: Herramientas para diseñar y modificar la estructura de las bases de datos.
   - **Manipulación de Datos**: Facilitan operaciones como inserción, actualización, borrado y consulta.
   - **Control de Acceso**: Gestionan la seguridad y los permisos de usuarios y roles.
   - **Respaldo y Recuperación**: Funcionalidades para proteger los datos y recuperarlos en caso de fallo.
   - **Integridad de Datos**: Aseguran la precisión y consistencia de los datos.
   - **Control de Concurrencia**: Permiten el acceso simultáneo de múltiples usuarios evitando conflictos.
4. **Arquitectura de un SGBD**:
   - Incluye componentes como el motor de base de datos, compilador de consultas y herramientas de administración.
   - Se divide en interfaz de usuario, middleware y almacenamiento de datos.
5. **Ejemplos de SGBD Comunes**:
   - **Relacionales**: MySQL, PostgreSQL, Oracle, SQL Server.
   - **No relacionales**: MongoDB, Cassandra, Redis.
6. **Consideraciones de Rendimiento**:
   - Métodos como índices y particionamiento para optimización.
   - Técnicas de tuning y monitoreo para mejorar el rendimiento de las bases de datos.

Los SGBD juegan un papel crucial en la administración eficiente de datos, proporcionando herramientas y funcionalidades para manejar grandes volúmenes de información de manera efectiva. Entender los diferentes tipos y funciones de los SGBD es fundamental para profesionales que trabajan con datos, ya que cada

sistema tiene sus propias características y se adapta a diferentes necesidades y entornos.

3.3.7 Bases de Datos Distribuidas

Las bases de datos distribuidas son sistemas donde los datos están almacenados y gestionados en múltiples ubicaciones físicas, a menudo repartidas geográficamente. Estos sistemas se conectan a través de una red, permitiendo el acceso y la gestión coherente y eficiente de los datos en todas las ubicaciones. Son fundamentales para organizaciones grandes y descentralizadas que requieren alta disponibilidad, escalabilidad y confiabilidad de sus datos.

Aspectos Clave de las Bases de Datos Distribuidas:

1. **Distribución de Datos**: Los datos se almacenan en diferentes sitios, cada uno con su propio sistema de gestión de bases de datos.
2. **Independencia de Ubicación**: Los usuarios pueden acceder a los datos sin necesidad de saber dónde están almacenados físicamente.
3. **Replicación**: Los datos se pueden replicar en múltiples sitios para mejorar el acceso y la redundancia.
4. **Fragmentación**: Los datos se pueden dividir (fragmentar) y distribuir en diferentes sitios para mejorar el rendimiento y la eficiencia local.
5. **Transparencia**: Los sistemas de bases de datos distribuidas proporcionan una apariencia unificada, de modo que la distribución de los datos es transparente para los usuarios.
6. **Sincronización y Consistencia**: Se mantienen mecanismos para asegurar que los datos en todos los sitios permanezcan sincronizados y consistentes.

Ejemplos de Aplicaciones de Bases de Datos Distribuidas:

1. **Sistemas Bancarios Globales**: Los bancos con múltiples sucursales en todo el mundo utilizan bases de datos distribuidas para mantener la coherencia de la información de las cuentas de los clientes, permitiendo transacciones en tiempo real en cualquier ubicación.
2. **Plataformas de Comercio Electrónico**: Empresas como Amazon utilizan bases de datos distribuidas para gestionar inventarios, pedidos de clientes y datos de usuarios en diferentes centros de datos alrededor del mundo.

3. **Redes Sociales**: Plataformas como Facebook y Twitter utilizan bases de datos distribuidas para almacenar y gestionar grandes cantidades de datos de usuarios, actualizaciones de estado y multimedia, asegurando un acceso rápido y confiable para usuarios en todo el mundo.
4. **Sistemas de Reservas de Viajes**: Las aerolíneas y agencias de viajes utilizan bases de datos distribuidas para gestionar reservas, horarios de vuelos y disponibilidad de asientos, proporcionando información actualizada y coherente a través de múltiples ubicaciones.

Estos ejemplos ilustran cómo las bases de datos distribuidas son esenciales en muchos sectores donde se requiere gestionar y acceder a grandes cantidades de datos de manera eficiente y confiable en un entorno globalizado.

3.4. Seguridad informática

Definición de Seguridad Informática:

- Conócete a ti mismo: Amenazas y vulnerabilidades.
- ¿Qué es un activo informático y su clasificación?
- Análisis de riesgos
- Identificar vulnerabilidades y amenazas, diferencia entre vulnerabilidad y amenaza
- Evaluación del riesgo (impacto – probabilidad)

Se pueden apoyar en la Sección 2 del libro de Natan House, The Complete Cyber Security Course.

Amenazas y vulnerabilidades:

- Tipos de amenazas
- Medidas para contrarrestar amenazas, plan de respuesta a riesgos
- Controles de acceso
- Mecanismos de seguridad para proteger la información:
    - de un sistema web
    - en plataformas móviles
    - en redes de computadoras

Se pueden apoyar en la Sección 3 del libro de Natan House, The Complete Cyber Security Course.

Para la seguridad en aplicaciones móviles, se sugiere dar un vistazo a:
https://www.nts-solutions.com/blog/seguridad-aplicaciones-moviles.html

Libro Seguridad en aplicaciones móviles de Marc Domingo Prieto:

Criptografía:

- Definición, historia y categorización
- Algoritmos de cifrado simétrico
  - Ventajas, en qué escenarios conviene utilizarse.
- Algoritmos de cifrado asimétrico
  - Ventajas, en qué escenarios conviene utilizarse.
- Autenticación y firmas digitales

Se pueden basar en el capítulo 7. Fundamentos de criptografía:
https://idoc.pub/documents/unidad-7-fundamentos-de-criptografia-ylyxqk05v3nm

Comparativa de algoritmo simétrico y asimétrico:
http://www.criptored.upm.es/descarga/Class4cryptc4c5.8_Comparativa_Cifra_Simetrica_Asimetrica.pdf

Firma digital:
https://www.youtube.com/watch?v=o8vBlQGhDdQ

3.4.1 Definición de Seguridad Informática

La seguridad informática, o ciberseguridad, es esencial en el mundo digital de hoy, donde la protección de la información y los sistemas informáticos es primordial para mantener la integridad, la confidencialidad y la disponibilidad de los datos. Los conceptos clave en seguridad informática incluyen:

1. **Conócete a ti mismo: Amenazas y Vulnerabilidades**:
   - **Amenazas**: Potenciales causas de incidentes perjudiciales, como ataques de malware, hacking, phishing, entre otros.
   - **Vulnerabilidades**: Debilidades en los sistemas que pueden ser explotadas por las amenazas. Ejemplos incluyen software desactualizado, configuraciones incorrectas, o debilidades en los procedimientos de seguridad.
2. **Activo Informático y su Clasificación**:
   - **Activo Informático**: Elementos valiosos en el ámbito de TI, como datos críticos, infraestructura de red, aplicaciones y personal.
   - **Clasificación**: Los activos se clasifican en base a su valor y rol en la organización. Por ejemplo, los datos del cliente pueden ser clasificados como críticos debido a su sensibilidad.

3. **Análisis de Riesgos**:
    - Proceso de identificar, evaluar y priorizar los riesgos, considerando tanto la probabilidad de que ocurran como el impacto que tendrían en la organización.
4. **Identificar Vulnerabilidades y Amenazas, Diferencia entre Vulnerabilidad y Amenaza**:
    - **Vulnerabilidad**: Una falla o debilidad. Por ejemplo, un firewall mal configurado puede ser una vulnerabilidad.
    - **Amenaza**: Algo que puede explotar una vulnerabilidad, como un atacante externo que utiliza técnicas de ingeniería social.
    - **Diferencia**: La vulnerabilidad es la debilidad en sí, mientras que la amenaza es lo que se aprovecha de esa debilidad.

La evaluación del riesgo en el contexto de la seguridad informática es un proceso estructurado para entender y gestionar la probabilidad y el impacto de eventos negativos, como ataques cibernéticos o fallas de seguridad. La evaluación del riesgo se centra en dos componentes principales: el impacto y la probabilidad.

1. **Evaluación del Impacto**:
    - **Identificación del Impacto**: Determina qué daño o pérdida puede causar una amenaza si se materializa. Esto puede incluir pérdida financiera directa, daño a la reputación de la empresa, interrupción del negocio, pérdida de datos sensibles, entre otros.
    - **Medición del Impacto**: El impacto se puede medir en términos monetarios, operativos o cualitativos, y a menudo se clasifica en niveles como bajo, medio o alto. Por ejemplo, una violación de datos que expone información sensible del cliente puede clasificarse como de alto impacto debido a las implicaciones legales y de reputación.
2. **Evaluación de la Probabilidad**:
    - **Identificación de la Probabilidad**: Estima la frecuencia con la que se espera que ocurra una amenaza. Esto se basa en factores como el historial de ataques similares, la frecuencia de ataques dirigidos a tecnologías similares o la exposición general a amenazas.
    - **Medición de la Probabilidad**: La probabilidad también puede clasificarse como baja, media o alta. Por ejemplo, si un software es conocido por ser un objetivo frecuente de ataques cibernéticos, la probabilidad de un compromiso se consideraría alta.

3. **Cálculo del Riesgo**:
    - El riesgo general se calcula combinando el impacto y la probabilidad. Un método común es utilizar una matriz de riesgo, donde se cruza la probabilidad con el impacto para determinar el nivel de riesgo (bajo, medio, alto).
    - Por ejemplo, un riesgo con alto impacto pero baja probabilidad podría clasificarse como riesgo medio, mientras que un riesgo con alto impacto y alta probabilidad se clasificaría como riesgo alto.
4. **Priorización y Mitigación**:
    - Una vez evaluados los riesgos, se priorizan para determinar dónde enfocar los recursos para la mitigación.
    - Las estrategias de mitigación pueden incluir fortalecer las medidas de seguridad, implementar controles adicionales, o transferir el riesgo (por ejemplo, a través de un seguro).

La evaluación del riesgo es un componente esencial de un programa de seguridad informática eficaz, ya que permite a las organizaciones identificar y prepararse para los riesgos más críticos, asegurando así la protección de sus activos más valiosos.

Estos conceptos forman la base para desarrollar estrategias de seguridad robustas y eficaces en cualquier entorno de TI. La comprensión profunda de estas áreas es fundamental para prevenir, detectar y responder adecuadamente a los incidentes de seguridad, salvaguardando así los activos críticos de una organización.

3.4.2 Amenazas y vulnerabilidades

En la seguridad informática, las amenazas y vulnerabilidades son conceptos clave que se refieren a los peligros potenciales que pueden comprometer la integridad, confidencialidad y disponibilidad de los sistemas y datos. A continuación, se desarrollan cada uno de estos aspectos:

Tipos de Amenazas

- **Ataques de fuerza bruta**: Intentos de adivinar contraseñas mediante la prueba repetida de diferentes combinaciones.
- **Malware**: Software malicioso, incluyendo virus, gusanos, troyanos, ransomware, spyware, adware, etc.

- **Phishing**: Engaños para obtener información sensible como contraseñas o detalles de tarjetas de crédito.
- **DoS y DDoS**: Ataques que buscan hacer recursos o servicios inaccesibles.
- **MitM**: Intercepciones no autorizadas de comunicaciones.
- **Inyección SQL**: Introducción de código malicioso en consultas SQL.
- **Ataques de día cero**: Explotación de vulnerabilidades desconocidas en software o hardware.
- **Amenazas internas**: Daños causados por empleados, ya sean intencionales o accidentales.

Medidas para Contrarrestar Amenazas y Plan de Respuesta a Riesgos

- **Educación y Concientización**: Capacitación sobre riesgos y buenas prácticas.
- **Políticas de Seguridad**: Normativas para la gestión de seguridad.
- **Planes de Respuesta a Incidentes**: Protocolos de acción ante violaciones de seguridad.
- **Copias de Seguridad y Recuperación de Datos**: Para restaurar información perdida.

Controles de Acceso

- **Autenticación**: Verificación de identidad mediante contraseñas, tokens o biometría.
- **Autorización**: Definición de qué acciones puede realizar un usuario.
- **Contabilidad (Accountability)**: Rastreo de actividades de usuario.

Mecanismos de Seguridad para Proteger la Información

- **En Sistemas Web**:
  - **Certificados SSL/TLS**: Para cifrado de comunicaciones.
  - **WAF (Web Application Firewall)**: Protección contra ataques web.
  - **Hardening de servidores**: Reforzamiento de la seguridad del servidor.
- **En Plataformas Móviles**:
  - **Sandboxing**: Aislamiento de aplicaciones.
  - **Cifrado de datos**: Tanto en el dispositivo como en transmisión.

    - **Actualizaciones y parches regulares**: Para mantener la seguridad actualizada.
- **En Redes de Computadoras**:
    - **Firewalls**: Control de tráfico de red.
    - **IDS/IPS**: Sistemas de detección y prevención de intrusiones.
    - **Segmentación de Redes**: División y control del tráfico de red.
    - **VPN**: Canales seguros para comunicación remota.

Estas medidas y mecanismos son esenciales en un enfoque integral de seguridad, combinando aspectos técnicos, organizativos y de capacitación para proteger eficazmente contra una amplia gama de amenazas y vulnerabilidades.

3.4.3 Criptografía

La criptografía es una disciplina esencial en el campo de la seguridad de la información, proporcionando las herramientas necesarias para proteger la comunicación y los datos en un entorno digital. Vamos a explorar cada uno de los aspectos mencionados:

Historia y Categorización

- **Historia**: La criptografía se ha utilizado históricamente para proteger mensajes importantes, desde los jeroglíficos del antiguo Egipto hasta los códigos utilizados en guerras.
- **Categorización**: Se divide en criptografía simétrica y asimétrica (o de clave pública).

Algoritmos de Cifrado Simétrico

- **Definición**: Usan la misma clave para cifrar y descifrar mensajes.
- **Ventajas**:
    - Rápidos y eficientes en el procesamiento.
    - Ideales para cifrar grandes cantidades de datos.
- **Escenarios de Uso**:
    - Cifrado de datos almacenados (en reposo).
    - Comunicaciones seguras en redes confiables.

Aquí tienes algunos ejemplos prácticos de cómo se utilizan los algoritmos de cifrado simétrico y asimétrico en diferentes escenarios:

Ejemplos de Cifrado Simétrico

1. **Cifrado de Discos Duros o Almacenamiento en la Nube**:
   - Algoritmo: AES (Advanced Encryption Standard).
   - Uso: Cifrar los datos almacenados en un disco duro o en un servicio de almacenamiento en la nube. Por ejemplo, herramientas como BitLocker en Windows utilizan AES para cifrar todo el contenido del disco, protegiéndolo contra accesos no autorizados.
2. **Redes Privadas Virtuales (VPN)**:
   - Algoritmo: 3DES o AES.
   - Uso: Las VPNs utilizan cifrado simétrico para proteger el tráfico de datos entre el cliente y el servidor VPN. Esto asegura que la información transmitida a través de una red pública (como Internet) esté protegida contra escuchas indeseadas.
3. **Comunicaciones Internas de la Empresa**:
   - Algoritmo: Blowfish o AES.
   - Uso: Cifrar comunicaciones internas en una red empresarial, como el intercambio de archivos o mensajes dentro de la empresa, asegurando que solo el personal autorizado pueda acceder a la información.

Ejemplos de Cifrado Asimétrico

1. **Intercambio Seguro de Claves en SSL/TLS**:
   - Algoritmo: RSA o ECC.
   - Uso: Durante una conexión HTTPS, se utiliza cifrado asimétrico (como RSA) para intercambiar de manera segura una clave simétrica que luego se usa para cifrar el resto de la comunicación. Esto se realiza para proteger la información transmitida entre un navegador web y un servidor.
2. **Firmas Digitales en Documentos Electrónicos**:
   - Algoritmo: RSA o DSA.
   - Uso: Al firmar digitalmente un documento, por ejemplo, un PDF, se utiliza el cifrado asimétrico para crear una firma que verifica la autenticidad del documento y del firmante. Esto es común en contratos digitales y en la autenticación de software.

3. **Autenticación de Usuarios en Servicios Online**:
    - Algoritmo: RSA o ECC.
    - Uso: Para autenticar usuarios en servicios online, especialmente en sistemas que requieren una alta seguridad, como la banca en línea. Las claves asimétricas permiten que los usuarios demuestren su identidad de forma segura sin revelar información sensible.

En cada uno de estos ejemplos, el tipo de cifrado se elige según las necesidades específicas de seguridad, eficiencia y gestión de claves del escenario en cuestión. El cifrado simétrico es ideal para grandes volúmenes de datos debido a su eficiencia, mientras que el asimétrico es esencial para la seguridad en la transferencia de claves y la autenticación.

La criptografía moderna es fundamental para garantizar la seguridad en el ámbito digital, protegiendo contra una amplia gama de amenazas y permitiendo transacciones seguras y privadas en un mundo cada vez más conectado. La comprensión de estos conceptos es esencial para los profesionales de TI, la seguridad cibernética y cualquier persona interesada en proteger la información digital.